Wellesley College ◊ CS251 Programming Languages ◊ Spring, 2000

PROBLEM SET 3
Due Wednesday, March 8, 2000

Problem 0: Studying BINDEX

All of the problems on this problem set involve the BINDEX language discussed in class or extensions to this language. Before attempting the problems, you should study the code for the implementation of the BINDEX language, which can be found in the following directory on Nike:

```
/usr/users/cs251/download/bindex
```

Although there is nothing to turn in for this problem, the rest of the problems will be significantly easier once you understand how BINDEX works.

I have not yet implemented a parser for BINDEX, so the implementation cannot handle the non-S-expression concrete syntax for BINDEX that I have been using in class. Instead, you should use the fully parenthesized prefix S-expression concrete syntax indicated by the following examples:

Non-S-Expression Concrete Syntax	S-Expression Concrete Syntax

program (a, b) =
  bind c **as** +(a,b)
  in div(c,2)
	(program (a b)
	  (bind c (+ a b)
	  (div c 2)))

program (a, b, c) =
  *(**bind** d **as** *(a,c)
  in bind e **as** –(d,b)
  in div(*(b,d), +(e,a)),
  bind e **as bind** b **as** *(12,a)
  in –(b,c)
  in div(e,b)
)
	(program (a b c)
	  (* (bind d (* a c)
	  (bind e (– d b)
	  (div (* b d) (+ e a))))
	  (bind e (bind b (* 12 a)
	  (- b c))
	  (div e b))))

An advantage of the S-expression concrete syntax is that it can be given directly as input to Scheme functions when quoted. For example:

```
;; Run the averaging program on the inputs 3 and 8
> (env-run  '(program (a b)
                 (bind c (+ a b)
                   (div c 2)))
            '(3 8))
5
```

In this representation, since they are just S-expressions, BINDEX programs can be named so that they are easily reusable:

```
> (define avg
    '(program (a b)
       (bind c (+ a b)
         (div c 2))))

> (env-run avg '(3 8))
5
```

```
> (env-run avg '(20 10))
15
```

Of course, when manipulating such programs you should only use the abstract syntax operators! For example:

```
> (program-params avg)
(a b c)

> (bind? (program-body avg))
#t

> (bind-name (program-body avg))
c
```

Do **not** use any "raw" Scheme list operations to manipulate BINDEX programs! That is, you should not use `car` and `cdr` to access the components of a node, nor should you use `cons`, `list`, `append`, etc. to create nodes. (But you may use these operations to manipulate lists of formal parameters, lists of binding names and definitions, etc.)

It is recommended that you get a sense for how BINDEX works by experimenting with the BINDEX system. To load the BINDEX system, evaluate the following expressions in MIT-Scheme:

```
(cd "/usr/users/cs251/download/bindex") ; Connect to BINDEX directory
(load "load-bindex.scm")
(cd "~") ; Return to your home directory
```

You can use `subst-run` to run the substitution model interpreter, and `env-run` to run the evaluation model interpreter. E.g.:

```
;; Using the avg program defined above
> (subst-run avg '(3 8))
5

;; Using the avg program defined above
> (env-run avg '(3 8))
5
```

When using `subst-run` it is helpful to trace the `subst` and `subst-eval` functions to get a sense for how computation proceeds. For example:

```
> (trace subst-eval)
subst-eval

> (trace subst)
subst

> (subst-run avg '(3 8))
Entry (subst 8 'b '(bind c (+ a b) (div c 2)))
|Entry (subst 8 'b '(+ a b))
| Entry (subst 8 'b 'a)
| ==> a
| Entry (subst 8 'b 'b)
| ==> 8
|==> (+ a 8)
|Entry (subst 8 'b '(div c 2))
| Entry (subst 8 'b 'c)
| ==> c
| Entry (subst 8 'b 2)
```

```
| ==> 2
|==> (div c 2)
==> (bind c (+ a 8) (div c 2))
Entry (subst 3 'a '(bind c (+ a 8) (div c 2)))
|Entry (subst 3 'a '(+ a 8))
| Entry (subst 3 'a 'a)
| ==> 3
| Entry (subst 3 'a 8)
| ==> 8
|==> (+ 3 8)
|Entry (subst 3 'a '(div c 2))
| Entry (subst 3 'a 'c)
| ==> c
| Entry (subst 3 'a 2)
| ==> 2
|==> (div c 2)
==> (bind c (+ 3 8) (div c 2))
Entry (subst-eval '(bind c (+ 3 8) (div c 2)))
|Entry (subst-eval '(+ 3 8))
| Entry (subst-eval 3)
| ==> 3
| Entry (subst-eval 8)
| ==> 8
|==> 11
|Entry (subst 11 'c '(div c 2))
| Entry (subst 11 'c 'c)
| ==> 11
| Entry (subst 11 'c 2)
| ==> 2
|==> (div 11 2)
|Entry (subst-eval '(div 11 2))
| Entry (subst-eval 11)
| ==> 11
| Entry (subst-eval 2)
| ==> 2
|==> 5
==> 5
5
```

Simlarly, when using `eval-run` it is helpful to trace the `env-eval` function to get a sense for how computation proceeds. For example:

```
> (trace env-eval)
env-eval

> (env-run avg '(3 8))
Entry (env-eval '(bind c (+ a b) (div c 2)) '((a 3) (b 8)))
|Entry (env-eval '(+ a b) '((a 3) (b 8)))
| Entry (env-eval 'a '((a 3) (b 8)))
| ==> 3
| Entry (env-eval 'b '((a 3) (b 8)))
| ==> 8
|==> 11
|Entry (env-eval '(div c 2) '((c 11) (a 3) (b 8)))
| Entry (env-eval 'c '((c 11) (a 3) (b 8)))
| ==> 11
| Entry (env-eval 2 '((c 11) (a 3) (b 8)))
| ==> 2
|==> 5
==> 5
5
```

The file `bindex-examples.scm` contains a few simple programs to experiment with, but you are encouraged to write some of your own as well. The `bindex-examples.scm` file also contains an implementation of a simple test suite that will test an evaluator on a list of programs and examples and compare the results to the expected results. The variable `test-suite` contains the test cases; it is initially defined to be as follows:

```
(define test-suite
  (list
   (list 'inc inc '(3) 4)
   (list 'c2f c2f '(100) 212)
   (list 'c2f c2f '(0) 32)
   (list 'c2f c2f '(-40) -40)
   (list 'calc calc '(20) 10)
   (list 'calc calc '(31) 15)
   (list 'avg avg '(3 8) 5)
   (list 'avg avg '(20 10) 15)
   (list 'test-bindseq test-bindseq '(3) 63)
   (list 'test-bindpar test-bindpar '(3) 37)
   (list 'test-bindseq2 test-bindseq2 '(3) 6)
   (list 'test-bindpar2 test-bindpar2 '(3) 149)
   (list 'test-ast test-ast '(5 4 2) 42)
   ))
```

You are encourage to add new programs and test cases to your own local copy of `bindex-examples.scm`.

The `test` function is used to test a program evaluator (such as `subst-run` or `env-run`) on the test cases in test-suite. For example:

```
> (test subst-run)

Running inc on (3) gives 4. OK!
Running c2f on (100) gives 212. OK!
Running c2f on (0) gives 32. OK!
Running c2f on (-40) gives -40. OK!
Running calc on (20) gives 10. OK!
Running calc on (31) gives 15. OK!
Running avg on (3 8) gives 5. OK!
Running avg on (20 10) gives 15. OK!
Running test-bindseq on (3) gives 63. OK!
Running test-bindpar on (3) gives 37. OK!
Running test-bindseq2 on (3) gives 6. OK!
Running test-bindpar2 on (3) gives 149. OK!
Running test-ast on (5 4 2) gives 42. OK!
Done.
#f
```

The `test` function complains if the actual result does not match the expected result specified in test-suite. For instance, if we change the line

```
   (list 'avg avg '(3 8) 5)
```

in `test-suite` to instead be the (incorrect) line

```
   (list 'avg avg '(3 8) 6)
```

then running `(test subst-run)` would indicate what it thinks is an error as follows:

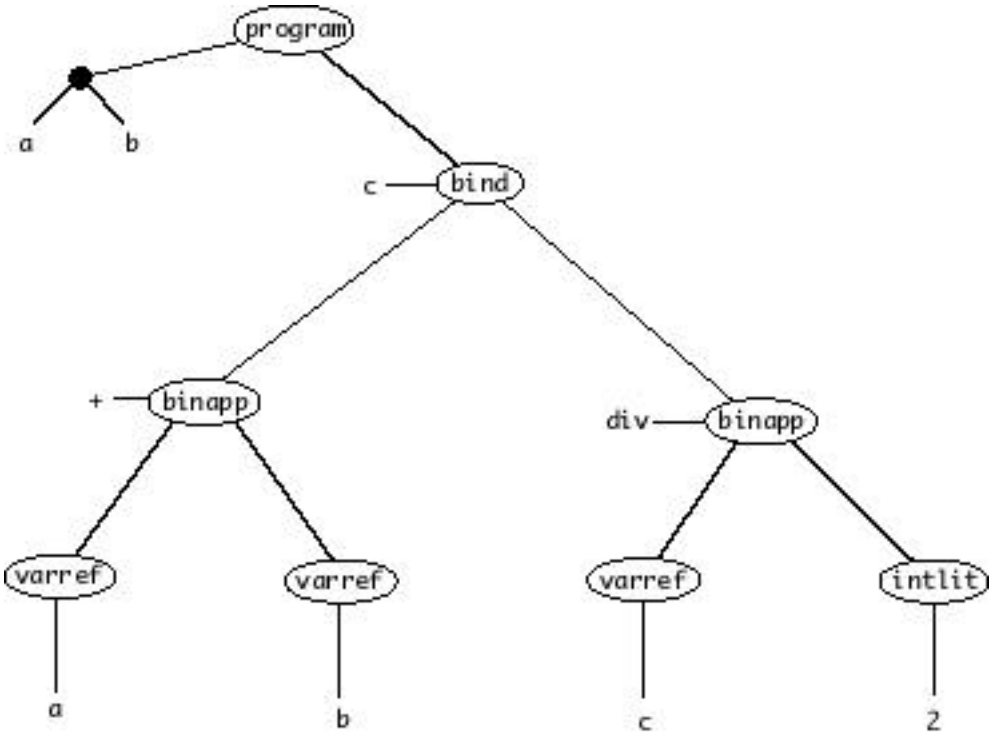
```
Running avg on (3 8) gives 5 ***ERROR!*** Expected 6
```

Problem 1 [25]: Abstract Syntax Trees

Consider the following BINDEX averaging program:

```
(program (a b)
  (bind c (+ a b)
    (div c 2)))
```

Here is the abstract syntax tree (AST) for this program:

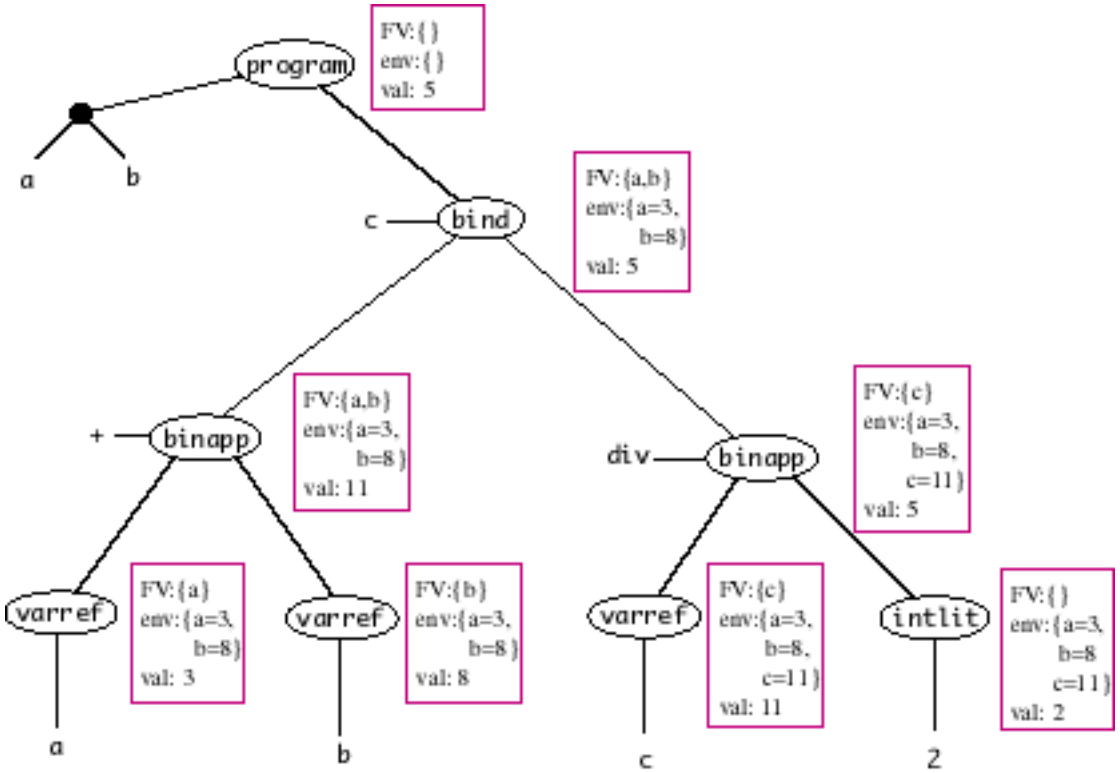


Note that the multiple parameters of the program are shown branching off a single solid node that stands for the sequenence of parameters.

Suppose we annotate each node of the abstract syntax tree with the following three pieces of information:

- The free variables of the program or expression rooted at the node.
- The environment in which the node would be evaluated if the program were run on the actual parameters a = 3 and b = 8. (Write environments as sets of bindings of the form *key* = *value*.)
- The value that would result from evaluating the node in the environment from (2).

The following picture shows the AST for the averaging program annotated with this information:



In this problem, you are to draw a similar annotated AST for the following BINDEX program:

```
(program (a b c) =
  (* (bind d as (* a c)
       (bind e as (– d b)
         (div (* b d) (+ e a))))
     (bind e (bind b (* 12 a)
                 (- b c))
       (div e b))))
```

You should annotate each node of the abstract syntax tree with the following three pieces of information:

- The free variables of the program or expression rooted at the node.
- The environment in which the node would be evaluated if the program were run on the actual parameters a = 5, b = 4, and c = 2. (Write environments as sets of bindings of the form *key* = *value*.)
- The value that would result from evaluating the node in the environment from (2).

Note: for this problem, you will need to use a very large sheet of paper and/or to write very small. It is strongly recommended that you write the solution using pencil (not pen) and paper. Don't waste your time attempting to format it on a computer with a drawing program.

Problem 2 [20]: Naming in BINDEX

Part a. Suppose that the following program is run on the argument 3. Indicate the value that each name will be bound to during the execution, and also indicate the resulting value of the program:

```
;; BINDSEQ test program
(program (x)
  (bind a (+ x 1)
    (bindseq ((a (* x a))
              (b (+ x a)))
      (bindseq ((a (- a b))
                (b (* a a)))
        (+ a b)))))
```

Part b. Redo part (a), except using a version of the program in which every `bindseq` has been replaced by `bindpar`.

```
;; BINDPAR test program
(program (x)
  (bind a (+ x 1)
    (bindpar ((a (* x a))
              (b (+ x a)))
      (bindpar ((a (- a b))
                (b (* a a)))
        (+ a b)))))
```

Part c. Write the result of desugaring the programs from both part a and part b into BINDEX programs that use only bind in place of `bindseq` and `bindpar`. (You should perform the desugaring by hand and not use Scheme to do it for you!)

Part d. In the `subst` function within the file `subst.scm`, the case for bind expression is as follows:

```
((bind? exp)
 (make-bind (bind-name exp)
            (subst int name (bind-defn exp))
              (if (eq? name (bind-name exp))
                  ;; Careful! Don't substitute in this case
                  (bind-body exp)
                  (subst int name (bind-body exp)))))
```

Explain why there is a special case to handle the situation where the bound variable of a bind expression is the same as the name being substituted away by the `subst` function. Use example(s) to illustrate what would go wrong if this case was not handled specially.

Problem 3 [25]: bind*

Suppose that BINDEX is extended with a new binding construct `bind*` that has the same syntax as `bindpar` and `bindseq` except that it uses the keyword `bind*` in place of `bindpar` and `bindseq`. Consider the following program using `bind*`:

```
(program (a b c)
  (bind* ((a (+ (* 10 b) c))
          (b (+ (* 10 a) c))
          (c (+ (* 10 a) b)))
    (+ a (+ b c))))
```

Of course, the result of the above program will depend on the semantics given to the `bind*` construct. Here is a skeleton of the definition of env-eval extended with a clause for `bind*`:

```
(define env-eval
  (lambda (exp env)
    (cond . . .
          ((bind*? exp)
           (env-eval (bind*-body exp)
                     (fold2 (lambda (name defn e)
                              (env-bind name
                                        (env-eval defn env1)
                                        env2))
                     env
                     (bind*-names exp)
                     (bind*-defns exp))))
           . . .)))
```

The meaning of the `bind*` construct depends on whether *`fold2`* is `foldl2` or `foldr2`, whether *`env1`* is `env` or `e`, and whether *`env2`* is `env` or `e`. For each of the following combinations, indicate the result of the above program on the input values ‘(1 2 3). Show your work for partial credit.

fold2	***env1***	***env2***
foldl2	e	e
foldl2	e	env
foldl2	env	e
foldl2	env	env
foldr2	e	e
foldr2	e	env
foldr2	env	e
foldr2	env	env

Problem 4 [50]: IBEX Evaluation

In this problem you will extend the BINDEX language to support boolean values and various constructs that manipulate boolean values. The resulting language is called IBEX, for Integer and Boolean Expression language. Before continuing with this problem, you should read the description of IBEX in Appendix A.

The implementation of BINDEX has been changed in a few ways to better support the implementation of IBEX. These changes are summarized in Appendix B, which you should also read before attempting this problem.

The code you will extend for this problem can be found in the `ibex` directory within the CS251 download directory. You should copy the files from this directory to your own local directory, where you will be changing them. To load the IBEX system, you should load the `load-ibex.scm` file by executing (load "load-ibex.scm") in MIT-Scheme. (You may need to use cd to first change the directory to the directory containing your local copy of the `ibex` directory.)

For this problem, you should turn in your final copies of the following files, all of which you will modify as part of this problem.

```
desugar.scm    env-eval.scm      free-vars.scm     primops.scm
rename.scm     subst-eval.scm    subst.scm
```

Part 4a [10]: Implementing IBEX Primops

You should begin by implementing the new IBEX primops, which consist of the relational primops (`<`, `<=`, `=`, `!=` , `>=`, `>`) and the logical primops (`band`, `bor`, `not`). See Appendix A for the meaning of this primops, and Appendix B for details on how to add new primops.

All you need to do for this part is to extend the definition of `primop-env` in the file `primops.scm`. You should test your changes by evaluating the expression

```
(test-primops env-run),
```

which will apply `env-run` to each program in a suite of test cases and compare the actual answer to the expected answer. If the two match, you will see an `OK!` at the end of a line. If the expected and actual answers for a test case do not match, you will see the string `***WRONG ANSWER!***` along with an explanation of the mismatch between the actual and expected answers.

Make sure you remember to re-evaluate the definition of `primop-env` every time you change it before you test it. Or better yet, evaluate `(load "load-ibex.scm")` before performing tests after making a change.

To see more about the test programs in the test suite, see the file `ibex-examples.scm`. (Feel free to extend the suites of test cases to include test cases of your own!) Each entry of `test-suite` is a list of the following four elements:

- the name of the tested program.
- an expression evaluating to the S-expression for the program.
- A list of arguments on which to test the program.
- The expected value of applyin the program to the arguments.

For example, if avg is defined as in Problem 0, then here is a sample test case:

```
(list 'avg avg '(3 8) 5)
```

When such a test case appears in the test suite, testing it will print the following:

```
Running avg on (3 8) gives 5. OK!
```

On the other hand, suppose the + primop was accidentally given the meaning of *. Then testing the above test case would yield:

```
Running avg on (3 8) gives 12 ***WRONG ANSWER!*** Expected 5
```

Part 4b [15] Implementing `if`, `scand`, and `scor` in the Environment Model

The next task is to extend the environment model interpreter (as embodied in the `env-eval` function in the file `env-eval.scm`) to handle the three constructs `if`, `scand`, and `scor`, as described in Appendix A. You should add one clause for each of these constructs to the main `cond` expression of `env-eval`.

You should test that these constructs work by evaluating the expression

```
(test-conditionals env-run-no-desugar)
```

This will evaluate a number of test cases involving `if`, `scand`, and `scor`. Your evaluation clauses are probably correct if all the test cases end in `OK!`.

By the way, the `env-run-no-desugar` is a version of `env-run` that does not desugar (see Part 4c) the program body before evaluating it. The `env-run` function, which does desugar the program body, will give errors if used in this context --- at least until the `desugar` function is extended to handle `if`, `scand`, and `scor` in Problem 4c.

Part 4c [15]: Desugaring `scand` and `scor`

It would be possible to complete the IBEX implementation by adding three clauses (one for `if`, one for `scand`, one for `scor`) to several other function definitions (e.g. `subst`, `subst-eval`, `free-vars`, etc.) But a better approach is to observe that `scand` and `scor` can be "desugared" into instances of if as follows:

`(scand` *`E1 E2`*`)` desugars to `(if` *`E1 E2`* `#f)`
`(scor` *`E1 E2`*`)` desugars to `(if` *`E1`* `#t` *`E2`*`)`

This means that any program with `scand` and `scor` can be desugared (i.e., rewritten) to a program that replaces every occurrence of `scand` and `scor` by if. If every IBEX program is first desugared before any further processing, then `scand` and `scor` do not have to be handled by any functions other than the `desugar` function that implements the desugaring. This greatly simplifies the rest of the language implementation, which only needs to handle the `if` construct. This strategy is used in BINDEX to simplify the handling of `bindseq` and `bindpar`, both of which can be desugared into `bind`.

In this part, you should implement the desugaring of `scand` and `scor` by:

- Adding clauses for `if`, `scand`, and `scor` to the `desugar` function in `desugar.scm`. The `desugar` function takes a single expression and returns the desugared version of that expression.
- Adding a clause for `if` to the `free-vars` function in `free-vars.scm`. The `free-vars` function returns a set of the free variables in an expression. For this, you will need the set operations described in class.
- Adding a clause for `if` to the `rename` function in `rename.scm`. The call `(rename` *`old new exp`*`)` renames all free occurrence of the variable *`old`* to the variable *`new`* in the expression *`exp`*. It may need to perform other renamings to prevent inadvertent variable capture.

Note that it is necessary to modify `free-vars` and `rename` at the same time as `desugar`, because the desugaring of `bindpar` depends on these functions. However, because both `free-vars` and `rename` are guaranteed to be called only on desugared expressions, they only need a clause for `if` (and not clauses for `scand` and `scor`).

After you make these changes, the `bindseq`, `bindpar`, `scand`, and `scor` clauses in `env-eval` will no longer be used when calling `env-run`, because all instances of these constructs will have been desugared away. However, these clauses will still be used when `env-eval-no-desugar` is invoked.

You should test your modifications by evaluating the expression

```
(test-ibex env-run)
```

This will evaluate a number of test cases involving `if`, `scand`, and `scor`, where programs are first desugared before evaluation. Your evaluation clauses are probably correct if all the test cases end in `OK!`.

Part 4d: Implementing `if` in the Substitution Model

As the final step of implementing IBEX, extend the `subst` function in `subst.scm` and `the subst-eval` function in `subst-eval.scm` to handle the `if` construct. Test your modification via

```
(test-ibex subst-run)
```

Problem 5 [30] Dynamic Type-Checking in IBEX

Informally, a type is a collection of values. We shall use the type `int` to stand for the collection of all integer values and the type `bool` to stand for the collection whose elements are true and false. The collections that these types stand for are disjoint, so no value can have more than one of these types. Furthermore, since every IBEX value is either an integer or a boolean, it has exactly one of these two types. So we can classify IBEX values by the following function:

```
(define type-of
  (lambda (val)
    (cond ((integer? val) 'int)
          ((boolean? val) 'bool)
          (else (throw 'type-of-unknown-value val))
          )))
```

In the BINDEX language, every value was an integer and every primop was a binary operator with two integer arguments and an integer result. Because of this uniformity, not a lot can go wrong in the evaluation of a BINDEX expression. The only kind of errors we can encounter when running a BINDEX program are:

- An unbound variable, such as `y` in `(program (x) (+ x y))`
- An attempt to divide a number by 0, as in `(program (x) (div x 0))` or `(program (x) (mod x 0))`.

But in IBEX, many additional kinds of problems can arise:

- A primop may get the wrong number of arguments – e.g., `(band (< 1 2))`, `(not (< 1 2) (< 3 4))`, `(+ 1 2 3)`.
- A primop may get one or more arguments that are the wrong type – e.g., `(+ 1 #t)`, `(band #f 3)`, `(not 3)`.
- An `if` expression may have a non-boolean test – e.g., `(if 2 (+ 3 4) (* 5 6))`.
- A `scand` or `scor` expression may have non-boolean rands. E.g. `(scand #t 3)`, `(scor #f 5)`.

All of these new kinds of errors are called **type errors**.

A language is said to have **dynamic type checking** if all type errors encountered at run-time (i.e., when the program is run) are reported. Scheme is an example of a language with dynamic type checking. For instance, evaluating `(+ 1 #t)` in Scheme generates an error

```
*** ERROR -- NUMBER expected: (+ 1 #t)
```

And evaluating (not 2 3) in Scheme generates an error like:

```
*** ERROR -- Wrong number of arguments passed to procedure: (not 2 3)
```

But Scheme embodies a different notion of type errors than IBEX. For instance, in Scheme, the addition procedure can have any number of arguments. So `(+ 1 2 3)` is sensible in Scheme, but not in IBEX. As another difference, Scheme treats any non-false test value in an if expression as true, so `(if 2 (+ 3 4) (* 5 6))` evaluates to 7 in Scheme. However, in IBEX, this expression should yield a type error. In Scheme, the `or` special form returns the first non-false value, so `(or 1 2)` returns 1. In contrast, the IBEX expression `(scor 1 2)` should generate a type error.

The implementations of the environment and substitution model interpreters from Problem 4 catch some type errors but not others. For instance, running the program `(primop (x) (+ x #t))` on the argument list '(1) will eventually attempt to apply the Scheme addition function to the arguments 1 and #t; this will generate a Scheme type error, as noted above. But unless you have been very careful in your implementation of Problem 4, it is likely that many IBEX programs that should generate type errors do not. For example, the following IBEX programs should all generate type errors --- do they in your implementation?

```
(env-run '(program (a b c) (+ a b c)) '(1 2 3))

(env-run '(program (a b c d e f) (if (+ a b) (+ c d) (* e f)))
         '(1 2 3 4 5 6))

(env-run '(program (a b) (scor a b)) '(1 2))
```

The goal of this problem is to modify the environment and substitution model interpreters for IBEX to report **all** dynamic type errors. To do this, you will need to modify the following three functions (and potentially add some new auxiliary functions as well):

- `primapply` in `primops.scm`
- `env-eval` in `env-eval.scm`
- `subst-eval` in `subst-eval.scm`

Upon encountering a dynamic type error, you should use the `throw` construct (see Appendix B) to generate an exception. (Do **not** use Scheme's `error` construct!) The throw construct takes two operands: a symbolic tag (i.e., a symbol) indicating the kind of exception, and an information value given more details about the particular exception. For IBEX dynamic type errors, you should use the following symbolic tags and values:

Kind of Type Error	Symbolic Tag	Information Value
Number of actual arguments does not match expected number of arguments.	`primop-wrong-number-of-args`	`(expected` *`expected-num`* `got` *`actual-num`* `in` *`offending-primapp`*`)`
The type of an actual argument does not match the expected type of the argument.	`primop-wrong-arg-type`	`(expected` *`expected-type`* `got` *`actual-value`* `in` *`offending-primapp`*`)`
An `if` expression has a non-boolean test.	`non-boolean-if-test`	The offending `test` value

Here are some examples of how the dynamic type errors would appear using env-run:

```
> (env-run '(program (a b c) (+ a b c)) '(1 2 3))
*** ERROR -- IBEX exception primop-wrong-number-of-args:  (expected 2 got 3
in (+ 1 2 3))

:> (env-run '(program (a b c) (+ a (< b c))) '(1 2 3))
*** ERROR -- IBEX exception primop-wrong-arg-type:  (expected int got #t in
(+ 1 #t))

> (env-run '(program (a b c d e f) (if (+ a b) (+ c d) (* e f)))
           '(1 2 3 4 5 6))
*** ERROR -- IBEX exception non-boolean-if-test: 3
```

You can test your dynamic type checking by evaluating the expressions

```
(test-dynamic env-run)

(test-dynamic subst-run)
```

These will run the interpreters on test suites containing various type errors, and will compare the results of your interpreter to the expected results.

Notes:

- For catching the wrong number of arguments error or the wrong type of argument error, you need to know the expected number and types of the arguments to each primop. This information is already available in the primop descriptor associated with each primop in primop-env. See Appendix B for details. For instance:

```
> (pdesc-num-args (env-lookup '+ primop-env))
2

> (pdesc-arg-types (env-lookup '+ primop-env))
(int int)

> (pdesc-result-type (env-lookup '+ primop-env))
int

> (pdesc-num-args (env-lookup 'not primop-env))
1

> (pdesc-arg-types (env-lookup 'not primop-env))
(bool)

> (pdesc-result-type (env-lookup 'not primop-env))
bool

> (pdesc-num-args (env-lookup '< primop-env))
2

> (pdesc-arg-types (env-lookup '< primop-env))
(int int)

> (pdesc-result-type (env-lookup '< primop-env))
bool
```

- The type errors for `scand` and `scor` do not have to be handled specially, because they will desugar to type errors in the corresponding if expression.

- The following function (already defined in `type-check.scm`) is a handy way to test if a given value has a given type.

```
(define type-error?
  (lambda (type val)
    (not (eq? type (type-of val)))))
```

- Another kind of error is an unknown primitive function name. But it turns out that this error is caught in the desugarer rather than the evaluators.

```
> (env-run '(program (a b) (/ a b)) '(1 2))
*** ERROR -- IBEX exception desugar-unknown-expression:  (/ a b)
```

Part 6 [50]: Static Type Checking in IBEX

A disadvantage of dynamic type-checking is that type errors are not caught until the program is run on actual arguments. Yet, without knowing the actual arguments, we can often deduce that a program will have a type error. Finding type errors in program without running them on actual arguments is called **static type checking**.

Let's assume that all actual arguments to a program must be integers. Then we can reason that the program `(program (a b c) (+ a (< b c)))` has a type error as follows:

- Since `b` and `c` are ints, (< b c) is a bool.
- Since `a` is an int, (+ a (< b c)) will attempt to add an int to a bool, which is a type error.

The process of type checking in IBEX can be performed in a manner similar to evaluation via the environment model interpreter. In analogy with `env-eval`, we can write a function `type-eval` that takes two arguments: (1) an expression to be type-checked and (2) an environment that binds names to types (rather than to values). If the expression is well-typed in the given environment (i.e., has no static type errors), the `type-eval` function will return the type of the given expression (i.e., the symbol `int` or the symbol `bool`). But if the expression contains a static type error, `type-eval` will throw an exception indicating the kind of type error encountered.

Using `type-eval`, it is easy to write a function `(type-check` *`program`*`)` that returns the type of the value computed by a program, assuming that all of its parameters are integers.

The kinds of static type errors includes the three dynamic type errors from Problem 5, plus two additional ones:

Kind of Type Error	Symbolic Tag	Information Value
Number of actual arguments does not match expected number of arguments.	`primop-wrong-number-of-args`	`(expected` *`expected-num`* `got` *`actual-num`* `in` *`offending-primapp`*`)`
The type of an actual argument does not match the expected type of the argument.	`primop-wrong-arg-type`	`(expected` *`expected-type`* `got` *`actual-type`* `in` *`offending-primapp`*`)`
An `if` expression has a non-boolean test.	`non-boolean-if-test`	The offending test expression.
The types of the two branches of an if expression do not match	`if-branch-mismatch`	`(then:` *`then-expression`* `has-type` *`then-type`* `else:` *`else-expression`* `has-type` *`else-type`*`)`
Unbound variable error	`unbound-variable`	The name of the unbound variable.

Note that it is considered a static type error for the two branches of an if to have different types even though an expression with such an "error" may not generate a dynamic type error. For example, `(+ 1 (if (< 1 2) (+ 3 4) (< 5 6)))` has a static type error but not a dynamic type error. In order to catch errors without running the program, we often have to add additional restrictions to our language that prohibit programs that would not give errors at run-time. In return, we get a guarantee that any program without a static type error will not generate a dynamic type error when it is executed.

For this problem, you are to flesh out the definitions of `type-check` and `type-eval` within the `file type-check.scm`. You should carefully study `env-run`, `env-eval`, and `primapply` before attempting this problem.

You can test your type-checker by evaluating the expression

```
(test-checker type-check)
```

which will run your type checker on a suite of sample programs (both with and without type errors).

Appendix A: IBEX Specification

The IBEX language is an extension to BINDEX that supports both Integer and Boolean EXpressions (hence, "IBEX"). Every BINDEX program is also a legal IBEX program. But additionally, IBEX supports the following features:

Boolean literals:: In addition to integer literal values, IBEX supports the boolean literal values true (written `#t`, as in Scheme) and false (written `#f`, as in Scheme).

Relational binary operators: In addition to the arithmetic binary operators (`+`, `-` , `*` , `div`, `mod`), IBEX supports the relational binary operators `<`, `<=`, `=`, `>=`, `>`, and `!=` (not equal). These expect two integer operands and return a boolean result. E.g. :

`(< 3 2)` evaluates to `#f`	`(<= 3 2)` evaluates to `#f`
`(< 3 3)` evaluates to `#f`	`(<= 3 3)` evaluates to `#t`
`(< 3 4)` evaluates to `#t`	`(<= 3 4)` evaluates to `#t`
`(= 3 2)` evaluates to `#f`	`(!= 3 2)` evaluates to `#t`
`(= 3 3)` evaluates to `#t`	`(!= 3 3)` evaluates to `#f`
`(= 3 4)` evaluates to `#f`	`(!= 3 4)` evaluates to `#t`
`(> 3 2)` evaluates to `#t`	`(>= 3 2)` evaluates to `#t`
`(> 3 3)` evaluates to `#f`	`(>= 3 3)` evaluates to `#t`
`(> 3 4)` evaluates to `#f`	`(>= 3 4)` evaluates to `#f`

If one of the arguments of an arithmetic or relational operator is a boolean instead of an integer, a "runtime type error" is generated. Such an error means that an attempt is being made to use one type of value in a context where another kind is expected.

Logical binary operators: : IBEX supports boolean conjunction (written `band`) and boolean disjunction (written `bor`) binary operators. These expect two boolean operands and return a boolean result.

`(band (< 1 2) (< 3 4))` evaluates to `#t`	`(bor (< 1 2) (< 3 4))` evaluates to `#t`
`(band (< 2 1) (< 3 4))` evaluates to `#f`	`(bor (< 2 1) (< 3 4))` evaluates to `#t`
`(band (< 1 2) (< 4 3))` evaluates to `#f`	`(bor (< 1 2) (< 4 3))` evaluates to `#t`
`(band (< 2 1) (< 4 3))` evaluates to `#f`	`(bor (< 2 1) (< 4 3))` evaluates to `#f`

These are **not** "short-circuit" operators – both operands are always evaluated. For example:

`(band (< (div 1 0) 2) (< 3 4))` raises an error
`(band (< (div 1 0) 2) (> 3 4))` raises an error
`(band (< 3 4) (< (div 1 0) 2))` raises an error
`(band (> 3 4) (< (div 1 0) 2))` raises an error

`(bor (< (div 1 0) 2) (< 3 4))` raises an error
`(bor (< (div 1 0) 2) (> 3 4))` raises an error
`(bor (< 3 4) (< (div 1 0) 2))` raises an error
`(bor (> 3 4) (< (div 1 0) 2))` raises an error

An attempt to use an integer in a logical operation generates a runtime type error. E.g., `(band #t 3)` and `(bor 7 #f)` generate such errors.

Logical unary operator: : IBEX supports a logical unary not operator:

`(not (< 2 3))` evaluates to `#f`
`(not (< 3 2))` evaluates to `#t`
`(not 3)` generates a runtime type error

Conditional expressions: IBEX supports a conditional expression with the same as in Scheme: `(if` *`test-exp then-exp else-exp`*`)`. As in Scheme, *`test-exp`* is first evaluated; if it is true, the result of evaluating *`then-exp`* is returned, and if it is false, the result of evaluating *`else-exp`* is returned. In either case, exactly one of *`then-exp`* and *`else-exp`* are evaluated.

```
(if (< 1 2) (+ 3 4) (* 5 6)) evaluates to 7
(if (< 1 2) (+ 3 4) (div 1 0)) evaluates to 7
(if (< 2 1) (+ 3 4) (* 5 6)) evaluates to 30
(if (< 2 1) (div 1 0) (* 5 6)) evaluates to 30
(+ (if (< 1 2) (+ 3 4) (* 5 6))
   (if (< 8 7) (* 9 10) (+ 11 12))) evaluates to 30
```

The value returned by an if may be either an integer or a boolean:

```
(if (< 1 2) (< 3 4) (< 6 5)) evaluates to #t
(if (< 1 2) (< 4 3) (< 5 6)) evaluates to #f
(if (> 1 2) (< 3 4) (< 6 5)) evaluates to #f
(if (> 1 2) (< 4 3) (< 5 6)) evaluates to #t

(if (< 1 2) (+ 3 4) (< 5 6)) evaluates to #7
(if (> 1 2) (+ 3 4) (< 5 6)) evaluates to #t
```

Unlike in Scheme, IBEX requires that the *`test-exp`* position evaluates to a boolean. (In contrast, Scheme treats any non-false value as true.)

`(if (- 1 2) (+ 3 4) (< 5 6))` generates a runtime type error

Short-circuit boolean constructs: IBEX supports "short-circuit" boolean conjunction (written `scand`) and boolean disjunction (written `scor`). These are similar to the logical binary operators `band` and `bor`, except that `scand` does not evaluate its second argument if its first is false, and `scor` does not evaluate its second argument if its first is true.

```
(scand (< 1 2) (< 3 4)) evaluates to #t   (scor (< 1 2) (< 3 4)) evaluates to #t
(scand (< 2 1) (< 3 4)) evaluates to #f   (scor (< 2 1) (< 3 4)) evaluates to #t
(scand (< 1 2) (< 4 3)) evaluates to #f   (scor (< 1 2) (< 4 3)) evaluates to #t
(scand (< 2 1) (< 4 3)) evaluates to #f   (scor (< 2 1) (< 4 3)) evaluates to #f

(scand (< (div 1 0) 2) (< 3 4)) raises an error
(scand (< (div 1 0) 2) (> 3 4)) raises an error
(scand (< 3 4) (< (div 1 0) 2)) raises an error
(scand (> 3 4) (< (div 1 0) 2)) evaluates to #f

(scor (< (div 1 0) 2) (< 3 4)) raises an error
(scor (< (div 1 0) 2) (> 3 4)) raises an error
(scor (< 3 4) (< (div 1 0) 2)) evaluates to #t
(scor (> 3 4) (< (div 1 0) 2)) raises an error
```

Appendix B: Changes to the BINDEX implementation to support IBEX

To implement the features of IBEX, you will need to extend an implementation of BINDEX. In order to simplify the implementation of IBEX, a few changes have been made to the implementation of BINDEX discussed in class.

Abstract Syntax

The abstract syntax for IBEX extends the abstract syntax for BINDEX. The BINDEX subset of IBEX abstract syntax is the same as discussed in class except for two changes:

1) Literals: The `make-intlit`, `intlit-value`, and `intlit?` have been replaced by `make-literal`, `literal-value`, and `literal?` In the BINDEX subset of IBEX, literal still means an integer, but in full IBEX literal means an integer or a boolean. This change allows the same code for handling literals in BINDEX to correctly handle them in IBEX. In the S-expression concrete syntax for IBEX, boolean literals are written as in Scheme (`#t` and `#f`). Use the Scheme procedures `integer?` and `boolean?` to test if a literal is an integer or boolean.

2) Primitive Applications: Although BINDEX supports only binary operators, IBEX supports a unary operator (`not`) as well. It is easy to imagine extending IBEX with other unary operators, as well as operators that have more than two operands. Rather than having separate abstract syntax for unary operator applications and binary operator applications, these have been combined into a more general *primitive application* form, which abstractly has two parts: (1) a *rator*, which is the name of the primitive operator ("primop" for short) and (2) the rands, which is the list of operand expressions for application. These are supported by the following abstract syntax functions:

`(make-primapp` *`rator rands`*`)`
Returns a primitive application node. *`rator`* should be a symbol denoting a primop. *`rands`* should be a list of expressions denoting the operands of the primitive application.

`(primapp-rator primapp)` Returns the rator of a primapp
`(primapp-rands primapp)` Returns the rands of a primapp
`(primapp? exp)` Returns true if exp is a primitive application node, otherwise false.

In the S-expression abstract syntax, the operand list is spliced into the parenthesized primitive application node – i.e., there is **not** an extra set of parentheses around the list of operands. For example, the primitive application `(- (+ 1 2) (* 3 4))` has – as its rator and a list of `(+ 1 2)` and `(* 3 4)` as its rands. The primitive application `(not (= 3 4))` has `not` as its rator and a singleton list containing `(= 3 4)` as its rands.

The abstract syntax for IBEX also includes the following operations for constructing, deconstructing, and testing for the conditional constructs (`if`, `scand`, `scor`):

```
(make-if test then-part else-part)
(if-test if-exp)
(if-then if-exp)
(if-else if-exp)
(if? exp)

(make-scand rand1 rand2)        (make-scor rand1 rand2)
(scand-rand1 scand-exp)         (scor-rand1 scand-exp)
(scand-rand2 scand-exp)         (scor-rand2 scand-exp)
(scand? exp)                    (scand? exp)
```

Environments

The environments used in IBEX are similar to those used in BINDEX. In particular, IBEX environment include the following operations discussed in class:

`(env-empty)`
Returns the empty environment.

`(env-bind` *`key value env`*`)`
Returns a new environment that has a binding between *`key`* and *`value`* in addition to all the bindings of *`env`*. The binding between *`key`* and *`value`* overrides any existing binding for *`key`* in *`env`*.

`(env-lookup` *`key env`*`)`
Returns the value bound to *`key`* in *`env`*, or a special unbound token if no such binding exists.

A difference between the env-lookup defined above and the one defined in class is that the one define above returns a distinguished "unbound" token (that is different from false) to indicate that a key is not bound in the environment. The unbound token is the only Scheme value for which the following predicate returns true:

`(unbound? value)`
Returns true if value is the distinguished unbound token and false otherwise.

In IBEX, it is necessary to distinguish the unbound token from the false value because the false value may be bound to a name in an environment. With this change, the variable reference clause of `env-eval` becomes:

```
((varref? exp)
 (let ((probe (env-lookup (varref-name exp) env)))
   (if (unbound? probe)
       (error "ENV-EVAL: Unbound variable -- "
              (list 'exp = exp 'env= env))
       probe)))
```

Additionally, there are more operations in the environment contract than discussed in class. See Appendix C for details.

Sets

The set operations are those presented in class, except for one addition: `union-map`, which unions together the sets that result from mapping a set-producing function over the elements of a list:

```
(define union-map
 (lambda (f lst)
   (foldr set-union
          (set-empty)
          (map f lst))))
```

Exception Handling

When a BINDEX interpreter encountered an error, it used Scheme's `error` construct to halt the computation and print a message indicating the problem. When it come to testing an interpreter on a suite of test programs, however, we don't want the interpreter to halt when it encounters an error. Rather, we would like to be able to observe the error, but proceed to evaluate other programs in the test suite.

In order to get this behavior, we need some sort of exception handling mechanism, like Java's `throw` and `try`. Recall that Java programs can use `throw` to "throw" an exception value that is "caught" by the nearest dynamically enclosing instance of `try`. Any pending computation between the point of the throw and the point of the try is aborted. (If you don't understand this, don't worry - we will study exception handling in more detail later in the semester.)

Although such an exception handling mechanism is not a part of standard Scheme, it is easy to implement one (in less than a page of code!). The mechanism uses the following `throw` and `catch` constructs:

> `(throw` *`tag info`*`)`
> Throw an exception with tag *`tag`* and information value *`info`* to the nearest dynamically enclosing `catch`, where it will be handled. All pending computation between the point of the throw and the point of the `catch` is aborted. It is assumed that the entire program is wrapped in a default exception handler that prints out the tag and info.
>
> `(catch` *`thunk handler`*`)`
> Assume that *`thunk`* is a zero-parameter procedure and handler is a two-argument procedure. Establish an exception handler *`handler`* that is in effect during the computation generated by applying *`thunk`* to zero arguments. If the computation does not throw any exceptions, then `catch` returns with the value of the computation. But if the computation throws an exception, `catch` returns the result of applying *`handler`* to the tag and info of the exception.

As an example, consider the following function and sample invocations:

```
(define catch-test
  (lambda (x)
    (catch (lambda ()
             (* (+ (if (even? x)
                       (throw 'even x)
                       x)
                   1)
                (- (if (< x 0)
                       (throw 'negative x)
                       x)
                   1)))
           (lambda (tag value) (list tag value)))))

> (catch-test 3)
8  ; = (+ (+ 3 1) (- 3 1)

> (catch-test 4)
(even 4)

> (catch-test -3)
(negative –3)
```

In the IBEX interpreter, you should always use `throw` rather than Scheme's `error` to indicate an error. The part of IBEX that tests suites of sample programs depends on this. Note: you should only have to put `throw` in your programs; you should not write any instances of `catch`.

Primitive Operators

During evaluation, a primitive application "works" only if the length of the rands list is the expected number of operands for the operator, and the types of the actual operands matches the types expected by the primitive operator. To keep track of this sort of information, we will create a primop descriptor (pdesc for short) for each primop that contains the following:

- The name of the primop.
- A list of the argument types expected by the primop. We will use the symbol int to represent the integer type and the symbol bool to represent the boolean type. The length of this list is the number of arguments of the primop.
- The result type of the primop.
- The Scheme function that computes the result of the primop for inputs of the right type.

The contract for to primop descriptors is as follows:

`(make-pdesc` *`name arg-types result-type scheme-function`*`)`
Returns a descriptor with the given information.

`(pdesc-num-args` *`descriptor`*`)`
Returns the number of arguments expected by the primop.

The following return the named component of the descriptor:
`(pdesc-name` *`descriptor`*`)`
`(pdesc-arg-types` *`descriptor`*`)`
`(pdesc-result-type` *`descriptor`*`)`
`(pdesc-function` *`descriptor`*`)`

For example, the descriptor for the addition primop can be created as follows:

```
(make-pdesc '+ '(int int) 'int +)
```

To help with the creation of binary arithmetic primops, we define the following auxiliary function:

```
(define make-binary-arithop
  (lambda (name function)
    (make-pdesc name '(int int) 'int function)))
```

We can collect all the primitive descriptors for a language into a primop environment that associates each primop name with its descriptor. For instance, below is the primop environment for BINDEX. This environment can easily be extended to add new primitive operations into a language. In this assignment you will see how to add relational and logical operators, but it would be straightforward to add operations for characters, strings, lists, arrays, etc.

```
(define primop-env
  (bindings->env
   (list
    ;; BINDEX primops
    (make-binding '+ (make-binary-arithop '+ +))
    (make-binding '- (make-binary-arithop '- -))
    (make-binding '* (make-binary-arithop '* *))
    (make-binding 'div (make-binary-arithop 'div quotient))
    (make-binding 'mod (make-binary-arithop 'mod remainder))
    )))
```

The key use of the primop environment is to define what it means to apply a primop to a list of values. Here is a `primapply` fuction that extends the functionality of the `binop-to-function` operator we studied in the context of BINDEX evaluation:

```
(define primapply
 (lambda (primop-name args)
   (let ((pdesc (env-lookup primop-name primop-env)))
     (if (unbound? pdesc)
         (error "PRIMAPPLY: Unknown primop -- " primop-name))
         (apply (pdesc-function pdesc) args)))))
```

For example, `(primapply '+ '(2 3))` yields `5`. The `primapply` function works for primops of any number of arguments, where the arguments can be of any type. It is implemented in terms of Scheme's primitive `apply` function, which applies a primitive to a list of values. The `primapply` function is used to define the `primapp` clause in `subst-eval` and `env-eval`:

```
;; PRIMAPPLY clause within SUBST-EVAL
((primapp? exp)
 (primapply (primapp-rator exp)
            (map subst-eval (primapp-rands exp))))

;; PRIMAPPLY clause within ENV-EVAL
 ((primapp? exp)
  (primapply (primapp-rator exp)
             (map (lambda (rand) (env-eval rand env))
                          (primapp-rands exp))))
```

Appendix C: Environment Contract

An environment is an immutable table-like data structure that maintains associations between keys and values. Several of the operations mention an explicit binding datatype, defined by the following contract:

`(make-binding` *`key value`*`)` Returns a binding between *key* and *value*.
`(binding-key` *`binding`*`)` Returns the key of *`binding`*.
`(binding-value` *`binding`*`)` Returns the value of *`binding`*.

There is also a distinguished unbound value used to indicate failure in `env-lookup`.

`(unbound?` *`value`*`)`
Returns true if *`value`* is the distinguished unbound token and false otherwise.

Here are the operations on environments:

`(bindings->env` *`bindings`*`)`
Returns the environment made from the given list of *`bindings`*.

`(env-bind` *`key value env`*`)`
Returns a new environment that has a binding between *`key`* and *`value`* in addition to all the bindings of *`env`*. The binding between *`key`* and *`value`* overrides any existing binding for *`key`* in *`env`*.

`(env->bindings env)`
Return a list of the bindings env. The resulting list should have at most one binding for any key The order of the names does not matter, but should be the for different calls to env-bindings on the same environment

`(env-empty)`
Returns the empty environment.

`(env-extend` *`keys values env`*`)`
Returns a new environment with bindings between *`keys`* and *`values`* in addition to all the bindings of *`env`*. These bindings override any existing bindings for *`keys`* in *`env`*.

`(env-lookup` *`key env`*`)`
Returns the value bound to *`key`* in *`env`*, or a special unbound token if no such binding exists.

`(env-keep` *`keys env`*`)`
Returns an environment containing only the bindings of *`env`* whose keys are in the list *`keys`*.

`(env-make` *`keys values`*`)`
Returns an environment containing bindings between the keys of the list *`keys`* and the corresponding values of the list *`values`*.

`(env-merge` *`env1 env2`*`)`
Returns a new environment that includes all the bindings of *`env1`* and all the bindings of *`env2`*. When a key is bound in both, the binding in *`env1`* takes precedence.

`(env-remove` *`keys env`*`)`
Returns an environment containing only the bindings of *`env`* whose keys are not in the list *`keys`*.

Problem Set Header Page:
Please make this the first page of your submission.

CS251 Problem Set 3
Due Saturday, March 11, 2000

Name:

Date & Time Submitted *(only if late)*:

Collaborators (*anyone you collaborated with in the process of doing the problem set)*:

*In the **Time** column, please estimate the time you spent on the parts of this problem set. Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	Time	Score
General Reading		
Problem 1 [25]		
Problem 2 [20]		
Problem 3 [25]		
Problem 4a [10]		
Problem 4b [15]		
Problem 4c [15]		
Problem 4d [10]		
Problem 5 [30]		
Problem 6 [50]		
Total [200]		